

Trabajo Práctico 2 — Introducción a caches

[6620] Organización de Computadoras
Segundo cuatrimestre de 2020

Alumno:	LONGO ELIA, Manuel
Alumno:	BURMAN, Federico
Alumno:	ZAIETZ, Azul

Índice

1. Objetivos	2
2. Alcance	2
3. Requisitos	2
4. Descripción	2
5. Desarrollo	2
5.1. Predicción teórica del comportamiento del cache	2
5.2. a) Detalle de todos los comandos usados para recolectar los datos de cada una de las corridas: invocacion de valgrind, cg annotate, etc.	3
5.3. b) Incorporar las anotaciones línea por línea de cg annotate correspondiente al archivo .c del benchmark analizado.	3
5.4. c) Explicar en detalle por qué se produce la cantidad de desaciertos indicada sobre L1D para cada una de las configuraciones del sistema de memoria del cuadro 1 1 .	5
5.5. d) Calcular la cantidad total de accesos a memoria, aciertos y desaciertos realizada por el cache L1D.	8
5.6. e) Calcular las tasa de desacierto para L1D asociada a cada combinacion de benchmark y configuracion del sistema de memoria.	8
5.7. f) Contrastar detalladamente los resultados de los calculos con las simulaciones. Justificar todo.	9
6. El código fuente de todos los archivos analizados	12
7. El código MIPS32 generado por el compilador	13
8. Conclusión	17
9. Link al repositorio de github	18

1. Objetivos

Estudiar el comportamiento de los sistemas de memoria cache utilizando una serie de escenarios de análisis o benchmarks descriptos a continuación.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 6, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

En este trabajo estudiaremos el comportamiento de una serie de configuraciones de sistemas de memoria cache, analizando la ejecución de un programa que permite multiplicar matrices cuadradas con `cachegrind` `cachegrind`.

A lo largo de este TP, adoptaremos las 3 configuraciones para el nivel 1 y 2 del sistema de memoria cache indicadas en el cuadro 1. En todos los casos la capacidad total será de 32 kbytes, y el tamaño de línea 32 bytes. Para cada una de estas configuraciones, deberá estudiarse el comportamiento de las mismas al ejecutar una implementación del algoritmo naive de multiplicación de matrices cuadradas. Para ello, deberá usarse el entorno QEMU del trabajo anterior [2], y la el programa `cachegrind` [3], una herramienta de profiling y simulación de sistemas de memoria cache multinivel que forma parte de la suite de software Valgrind [4].

5. Desarrollo

Usando las herramientas, deberá ejecutarse el algoritmo naive de multiplicación de matrices repetidas veces, con tamaño de matriz progresivamente creciece: $n = 0, 2, 4, 8, \dots$, en donde n representa la cantidad de filas y columnas de la matriz. Deberán estudiarse los accesos que el algoritmo realiza sobre el sistema de memoria para acceder a los elementos de la matriz y calcular los productos internos de los vectores fila y columna. Los resultados deberan representarse en un diagrama como el de la figura 1 que permita representar en la cantidad de desaciertos del cache L1D en funcion de n , como así tambien la tasa misses asociada. Asimismo, cada grupo deberá realizar una predicción teórica del comportamiento del cache, el cual deberá contrastarse con las mediciones de `cachegrind` y documentadas en el informe (coincidan o no). En este trabajo práctico no se requiere estudiar el comportamiento de otros niveles de cache (solo L1D).

5.1. Predicción teórica del comportamiento del cache

Las memorias caché a analizar tendrán diferentes comportamientos basados en sus estructuras físicas (compuertas), y tambien en sus métodos de alocaación de bloques de memoria en Cache. Estos diferentes comportamientos se verán reflejados en los tiempos de ejecución de los programas para los mismos benchmarks y también en la cantidad de misses y hits que se producen. Una de las cosas a analizar en el tp es que tanta relación hay entre el tiempo de ejecución y la cantidad de misses y hits que hay en cada tipo de caché. Las asociativas, a diferencia de las de mapeo directo, cuentan con un mayor grado de complejidad electrónica, lo que implica más cantidad de

multiplexores y comparaciones por hacer. Esto puede llevar a mayor tiempo de ejecución para misma cantidad de misses. Por otro lado, si la dimensión de las matrices aumenta, y también así la posibilidad de que se produzcan misses en caché, las asociativas tenderán a tener menos misses por su complejidad y su mayor flexibilidad para guardar diferentes bloques de memoria en distintos lugares de la caché.

5.2. a) Detalle de todos los comandos usados para recolectar los datos de cada una de las corridas: invocacion de valgrind, cg annotate, etc.

Para compilar:

```
cc -Wall -g -o /tmp/mult main.c matrix.c multiplicar.c
```

Para correr el cachegrind sobre el programa:

```
/opt/valgrind/bin/valgrind --I1=32768,4,32 --D1=32768,2,32
--tool=cachegrind /tmp/mult < input.txt
```

Para cada tipo de memoria, se corrió el parámetro característico indicado en la tabla del enunciado.

Para correr el Cg annotate sobre la función que multiplica las matrices usamos el siguiente comando especificándole cuál es el archivo generado por el cache grind:

```
/opt/valgrind/bin/cg_annotate cachegrind.out.1170 /root/TP2/OrgaCompusTP2/multiplicar.c
```

5.3. b) Incorporar las anotaciones línea por línea de cg annotate correspondiente al archivo .c del benchmark analizado.

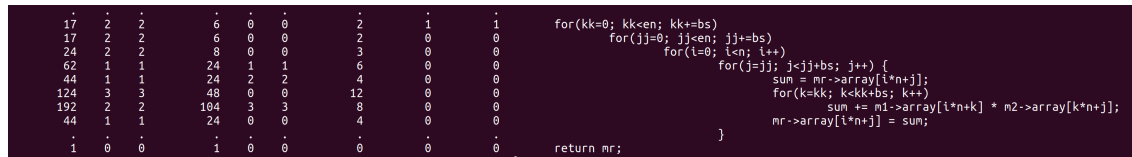


Figura 1: Matriz de 2x2 - Misses iguales para todos los tipos de caches



Figura 2: Matriz de 16x16 - Misses iguales para todos los tipos de caches

```

- - - - - - - - - - - #endif
- - - - - - - - - - -
17 2 2 6 0 0 2 1 1 for(kk=0; kk<en; kk+=bs)
17 2 2 6 0 0 2 0 0 for(jj=0; jj<en; jj+=bs)
520 2 2 194 0 0 65 0 0 for(i=0; i<n; i++)
41,664 1 1 16,640 9 1 4,160 0 0 for(j=jj; j<jj+bs; j++) {
45,056 1 1 24,576 1,046 1,026 4,096 4 0 sum = mr->array[i*n+j];
2,666,496 3 3 1,064,960 260 0 266,240 0 0 for(k=kk; k<kk+bs; k++)
6,291,456 2 2 3,407,872 17,691 2,049 262,144 0 0 sum += m1->array[i*n+k] * m2->array[k*n+j];
45,056 1 1 24,576 0 0 4,096 4,033 0 mr->array[i*n+j] = sum;
- - - - - - - - - - -
1 0 0 1 0 0 0 0 0 return mr;
6 0 0 2 0 0 0 0 0 }

```

Figura 3: Matriz 64x64 para C1

```

- - - - - - - - - - -
17 2 2 6 0 0 2 1 1 for(kk=0; kk<en; kk+=bs)
17 2 2 6 0 0 2 0 0 for(jj=0; jj<en; jj+=bs)
2,056 2 2 770 0 0 0 257 0 for(i=0; i<n; i++)
658,176 1 1 263,216 129 1 65,792 0 0 for(j=jj; j<jj+bs; j++) {
720,896 1 1 393,216 16,706 16,386 65,536 64 0 sum = mr->array[i*n+j];
168,493,056 3 3 67,371,008 16,448 0 16,842,752 0 0 for(k=kk; k<kk+bs; k++)
402,653,184 2 2 218,103,808 17,116,114 1,432,033 16,777,216 0 0 sum += m1->array[i*n+k] * m2->array[k
*n+j];
720,896 1 1 393,216 0 0 65,536 65,536 65,536 mr->array[i*n+j] = sum;
- - - - - - - - - - -
1 0 0 1 0 0 0 0 0 return mr;
6 0 0 2 0 0 0 0 0 }

```

Figura 4: Matriz de 256x256 para C1

```

- - - - - - - - - - - #endif
- - - - - - - - - - -
17 2 2 6 0 0 2 1 1 for(kk=0; kk<en; kk+=bs)
17 2 2 6 0 0 2 0 0 for(jj=0; jj<en; jj+=bs)
520 2 2 194 0 0 65 0 0 for(i=0; i<n; i++)
41,664 1 1 16,640 1 1 4,160 0 0 for(j=jj; j<jj+bs; j++) {
45,056 1 1 24,576 1,026 1,026 4,096 0 0 sum = mr->array[i*n+j];
2,666,496 3 3 1,064,960 0 0 266,240 0 0 for(k=kk; k<kk+bs; k++)
6,291,456 2 2 3,407,872 18,869 2,049 262,144 0 0 sum += m1->array[i*n+k] * m2->array[k*n+j];
45,056 1 1 24,576 0 0 4,096 3,080 0 mr->array[i*n+j] = sum;
- - - - - - - - - - -
1 0 0 1 0 0 0 0 0 return mr;
6 0 0 2 0 0 0 0 0 }

```

Figura 5: Matriz 64x64 para C2

```

- - - - - - - - - - -
17 2 2 6 0 0 2 1 1 for(kk=0; kk<en; kk+=bs)
17 2 2 6 0 0 2 0 0 for(jj=0; jj<en; jj+=bs)
2,056 2 2 770 0 0 0 257 0 for(i=0; i<n; i++)
658,176 1 1 263,216 1 1 65,792 0 0 for(j=jj; j<jj+bs; j++) {
720,896 1 1 393,216 16,386 16,386 65,536 0 0 sum = mr->array[i*n+j];
168,493,056 3 3 67,371,008 0 0 16,842,752 0 0 for(k=kk; k<kk+bs; k++)
402,653,184 2 2 218,103,808 16,874,562 1,398,788 16,777,216 0 0 sum += m1->array[i*n+k] * m2->array[k
*n+j];
720,896 1 1 393,216 0 0 65,536 65,536 65,536 mr->array[i*n+j] = sum;
- - - - - - - - - - -
1 0 0 1 0 0 0 0 0 return mr;
6 0 0 2 0 0 0 0 0 }

```

Figura 6: Matriz 256x256 para C2

```

- - - - - - - - - - - #endif
- - - - - - - - - - -
17 2 2 6 0 0 2 1 1 for(kk=0; kk<en; kk+=bs)
17 2 2 6 0 0 2 0 0 for(jj=0; jj<en; jj+=bs)
520 2 2 194 0 0 65 0 0 for(i=0; i<n; i++)
41,664 1 1 16,640 1 1 4,160 0 0 for(j=jj; j<jj+bs; j++) {
45,056 1 1 24,576 1,026 1,026 4,096 0 0 sum = mr->array[i*n+j];
2,666,496 3 3 1,064,960 0 0 266,240 0 0 for(k=kk; k<kk+bs; k++)
6,291,456 2 2 3,407,872 30,861 2,049 262,144 0 0 sum += m1->array[i*n+k] * m2->array[k*n+j];
45,056 1 1 24,576 0 0 4,096 3,072 0 mr->array[i*n+j] = sum;
- - - - - - - - - - -
1 0 0 1 0 0 0 0 0 return mr;
6 0 0 2 0 0 0 0 0 }

```

Figura 7: Matriz de 64x64 para C3

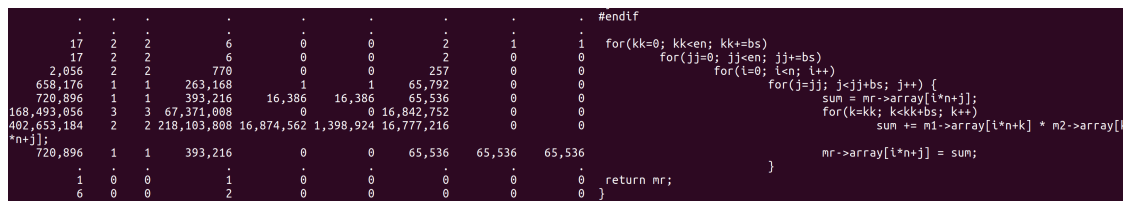


Figura 8: Matriz de 256x256 para C3

5.4. c) Explicar en detalle por qué se produce la cantidad de desaciertos indicada sobre L1D para cada una de las configuraciones del sistema de memoria del cuadro 1 1

- Acceso a instrucciones en memoria

La primera columna que genera el Cachegrind muestra la cantidad de accesos a instrucciones que hace el programa. La segunda columna muestra la cantidad de misses que se producen en la memoria cache cuando se quiere acceder a dichas instrucciones. Como era de esperar, la cantidad de accesos a las instrucciones aumenta de manera exponencial a medida que se aumenta la dimensión de las matrices a multiplicar. Sin embargo, la cantidad de misses que se producen es fija, sin importar el tipo de memoria caché ni la dimensión utilizada. Esto significa que una vez que se cargan en caché las instrucciones que multiplican las matrices, quedan almacenadas durante todo el proceso. Cada línea de C suele representar varias líneas de código en assembly, por lo que explica que por más que los misses ocurran sólo en la primera iteración de cada uno de los for, el número de misses por línea sea mayor a 1.

- Acceso a datos

En los accesos a los valores de las matrices es donde podemos encontrar las mayores diferencias en los comportamientos de las diferentes arquitecturas de caché.

De todas formas, estas diferencias no aplican para todos los benchmarks. Se puede observar que para las multiplicaciones de las matrices de 2x2 y las de 16x16, la cantidad de misses de caché es la misma. Esto se puede explicar por la simple razón de que a menor cantidad de accesos a datos, menos posibilidad tiene cualquier caché de tener un miss. Las matrices de 2x2 representan en memoria 32 bytes y las de 16, 2048 bytes, estos numeros son significativamente menores al tamaño de la memoria caché. Las matrices de 64x64 ocupan 16 KiloBytes, por lo que 2 de estas ocuparían toda la caché. Como esto no es posible ya que en la caché no solo hay variables de los programas corriendo, las matrices no se pueden almacenar todas juntas en la memoria cache, por lo que esto explica el leve aumento del miss rate para matrices de 64x64 y tan pronunciado para las de 256x256 (que ocupan 250 kilobytes aproximadamente en memoria cada una, el equivalente a 8 memorias cache de 32 Kb).

Para el caso de la multiplicación de matrices de grandes dimensiones, se puede ver en la columna D1mw de la salida de cg_annotate, que hay una alta cantidad de misses. Esa columna indica la cantidad de misses correspondientes a la lectura de datos. Como son matrices de gran tamaño, no llegan a almacenarse completamente en memoria cache. Para poder efectuar el producto matricial, el programa necesita acceder repetidas veces a los datos de las mismas y se generan muchos accesos a memoria (fallos de cache). Según la caché probada los valores de los misses varían. Como era de esperar, en la cache con mapeo directo ocurrieron más misses que en las caches asociativas por conjuntos, se puede observar también que en las C2 y C3 los misses de accesos a datos ocurrieron principalmente en 2 líneas, las que se acceden a los valores de las matrices, mientras que en la C1 los misses ocurren esas mismas 2 líneas sumadas a la que se hace la comparación con los índices que recorren las matrices. La capacidad asociativa de las cachés C2 y C3, disminuyeron la cantidad de misses en accesos a los valores de las matrices y llevaron a 0 los misses en accesos a los valores de

los índices que recorren las mismas.

■ Tiempo de ejecución

A continuación, se adjuntan imágenes de las corridas de distintos tamaños de matrices con cada uno de los tipos de memoria cache.

```

433 1303 1394 1433 1337 1477 1432 1708 1434 1434 1008 1470 1431 1303 1711 1473 1090 1407 1
308 1425 1322 1130 1452 1346 1383 1264 1286 1654 1244 1244 1260 1186 1370 1302 1302 1189 1
260 1167 1262 1369 1318 1516 1271 1203 1504 1391 1251 1360 1395 1228 1351 1328 1185 1241 1
time: 21.546
==351==
==351== I   refs:      173,453,914
==351== I1  misses:      2,544
==351== LLi misses:      2,517
==351== I1  miss rate:      0.00%
==351== LLi miss rate:      0.00%
==351==
==351== D   refs:      84,865,391 (61,163,415 rd + 23,701,976 wr)
==351== D1  misses:      1,389,263 ( 53,027 rd + 1,336,236 wr)
==351== LLd misses:      1,320,992 ( 6,095 rd + 1,314,897 wr)
==351== D1  miss rate:      1.6% ( 0.1% + 5.6% )
==351== LLd miss rate:      1.6% ( 0.0% + 5.5% )
==351==
==351== LL refs:      1,391,807 ( 55,571 rd + 1,336,236 wr)
==351== LL misses:      1,323,509 ( 8,612 rd + 1,314,897 wr)
==351== LL miss rate:      0.5% ( 0.0% + 5.5% )

```

Figura 9: Matriz de 64x64 para C1

```

308 1425 1322 1130 1452 1346 1383 1264 1286 1654 1244 1244 1260 1186 1370 1302 1302 1189 1
260 1167 1262 1369 1318 1516 1271 1203 1504 1391 1251 1360 1395 1228 1351 1328 1185 1241 1
time: 13.9883
==355==
==355== I   refs:      173,453,775
==355== I1  misses:      2,550
==355== LLi misses:      2,523
==355== I1  miss rate:      0.00%
==355== LLi miss rate:      0.00%
==355==
==355== D   refs:      84,865,322 (61,163,384 rd + 23,701,938 wr)
==355== D1  misses:      1,343,603 ( 25,483 rd + 1,318,120 wr)
==355== LLd misses:      1,320,992 ( 6,095 rd + 1,314,897 wr)
==355== D1  miss rate:      1.6% ( 0.0% + 5.6% )
==355== LLd miss rate:      1.6% ( 0.0% + 5.5% )
==355==
==355== LL refs:      1,346,153 ( 28,033 rd + 1,318,120 wr)
==355== LL misses:      1,323,515 ( 8,618 rd + 1,314,897 wr)
==355== LL miss rate:      0.5% ( 0.0% + 5.5% )

```

Figura 10: Matriz de 64x64 para C2

```

time: 15.9251
==357==
==357== I   refs:      173,453,722
==357== I1 misses:      2,544
==357== LLi misses:      2,517
==357== I1 miss rate:      0.00%
==357== LLi miss rate:      0.00%
==357==
==357== D   refs:      84,865,313 (61,163,377 rd + 23,701,936 wr)
==357== D1 misses:      1,355,306 ( 37,242 rd + 1,318,064 wr)
==357== LLd misses:      1,320,992 ( 6,095 rd + 1,314,897 wr)
==357== D1 miss rate:      1.6% ( 0.1% + 5.6% )
==357== LLd miss rate:      1.6% ( 0.0% + 5.5% )
==357==
==357== LL refs:      1,357,850 ( 39,786 rd + 1,318,064 wr)
==357== LL misses:      1,323,509 ( 8,612 rd + 1,314,897 wr)
==357== LL miss rate:      0.5% ( 0.0% + 5.5% )

```

Figura 11: Matriz de 64x64 para C3

Para hacer las comparaciones del tiempo de ejecución decidimos probar el benchmark de matrices de 64x64 por ser lo suficientemente grandes para que generen misses en los programas. Los resultados no fueron los que pensamos que ocurrirían, en la primera ejecución de la caché C1 el tiempo fue de 21 segundos, con la C2 el tiempo se acortó a 13 segundos y en C3 subió a 15. Luego de esto decidimos volver a correr los programas de vuelta para ver si podíamos ratificar estos tiempos o si eran una excepción. En las nuevas corridas, los tiempos cambiaron de vuelta, siendo de 9 segundos aproximadamente para cada caché.

Lo que podemos concluir de esto es que el tiempo de ejecución no depende únicamente de la caché que se usa, sino de factores externos a lo que ocurre en el programa, estos pueden ser procesos paralelos que están corriendo el procesador u otros que no sabemos.

■ Escritura de datos

Cuando observamos las columnas 7 y 8 que genera el `Cg_annotate` podemos ver los accesos a direcciones para escritura (columna 7) y los misses de caché para escritura (columna 8).

Para las matrices de 2x2 y 16x16, no hay ningún miss de escritura, esto se debe a que se están escribiendo en bloques de memoria que ya están cargados en la memoria caché. Las dos dimensiones ocupan poco espacio físico en memoria por lo que no es necesario escribir en otro bloque.

Como se puede ver en todas las figuras 3 a 8, en la línea que dice `"mr->array[i*n+j] = sum"` se producen casi el 100% de los misses de escritura con un miss rate del casi 100%. Esto se debe a que se está escribiendo en una variable completamente nueva, no se sobrescribe alguna de las matrices previamente usadas. Esta nueva matriz está en partes de memoria que no están en caché que ya está completamente ocupada por las matrices siendo multiplicadas, por lo que cada vez que se quiere escribir ocurre un miss. En todas las de 256x256 la cantidad de misses en la línea `"mr->array[i*n+j] = sum."` es de 65536 que es el resultado de la multiplicación 256x256, es decir la cantidad total de números que contiene la matriz calculada. En la escritura de las matrices de 64x64 es donde los misses de escritura más varían según el tipo de caché. En el caso de la caché C1 (figura 3) se produjeron 4033 misses del total de 4096 escrituras en memoria, lo que nos dice que en la caché de mapeo directo, hubo 63 valores que tuvieron un write hit. Para las 2 cachés asociativas por conjuntos anlizadas, los misses en escrituras en caché para matrices de 64x64 fueron bastante parecidas, teniendo un missrate del 75%. Estos valores similares se explican por su similar manera de funcionar.

Esto nos da una pauta de que las caches están funcionando con la política de escritura write allocate, es decir que tiene que escribir primero en cache para luego escribir en la memoria. Sin la política de escritura que tienen las caches fuera de write no allocate, no habría ocurrido ningún miss de escritura, es decir, se hubiera escrito directamente en la memoria principal sin pasar por la caché.

5.5. d) Calcular la cantidad total de accesos a memoria, aciertos y des- aciertos realizada por el cache L1D.

La tasa total de aciertos se puede calcular como la diferencia entre la cantidad de datos leídos de memoria y la cantidad de misses.

Benchmark	TIPO DE CACHE		
	C1	C2	C3
2	238	238	238
16	74682	74682	74682
64	4519891	4338934	4326942
256	269375737	269634367	269634239

Figura 12: Datos misses

5.6. e) Calcular las tasa de desacierto para L1D asociada a cada com- binacion de benchmark y configuracion del sistema de memoria.

Benchmark	Tipo de cache					
	C1		C2		C3	
	Misses de datos	Miss rate de datos	Misses de datos	Miss rate de datos	Misses de datos	Miss rate de datos
2	6	0.02459	6	0.02459	6	0.02459
16	196	0.00261	196	0.00261	196	0.00261
64	18939	0.00417	19896	0.00456	31888	0.00731
256	17149461	0.05985	16890821	0.05895	16890949	0.05895

Figura 13: Datos misses

5.7. f) Contrastar detalladamente los resultados de los calculos con las simulaciones. Justificar todo.

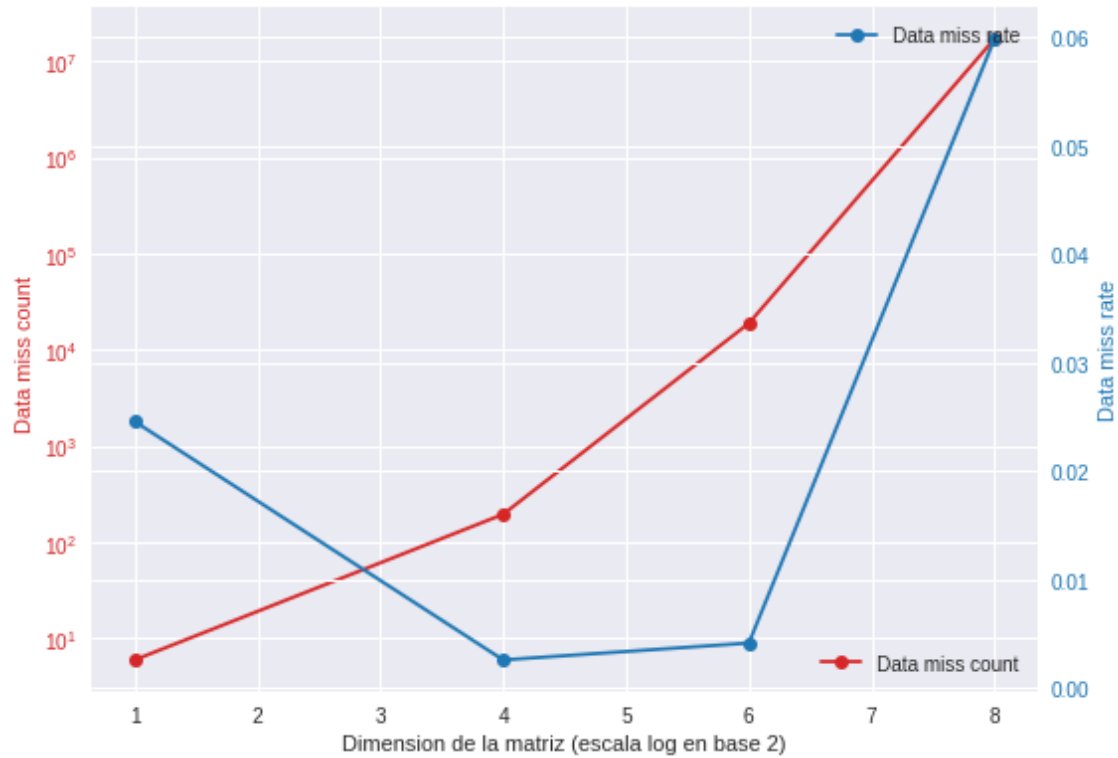


Figura 14: Misses de datos para C1

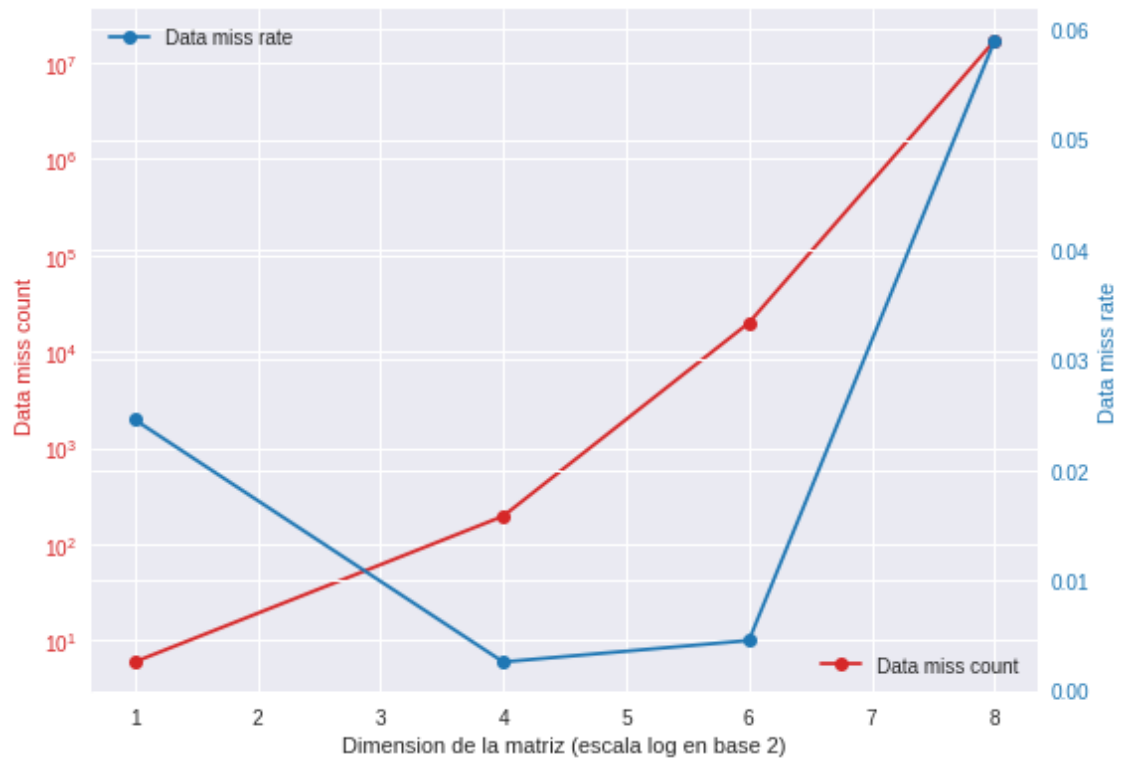


Figura 15: Misses de datos para C2

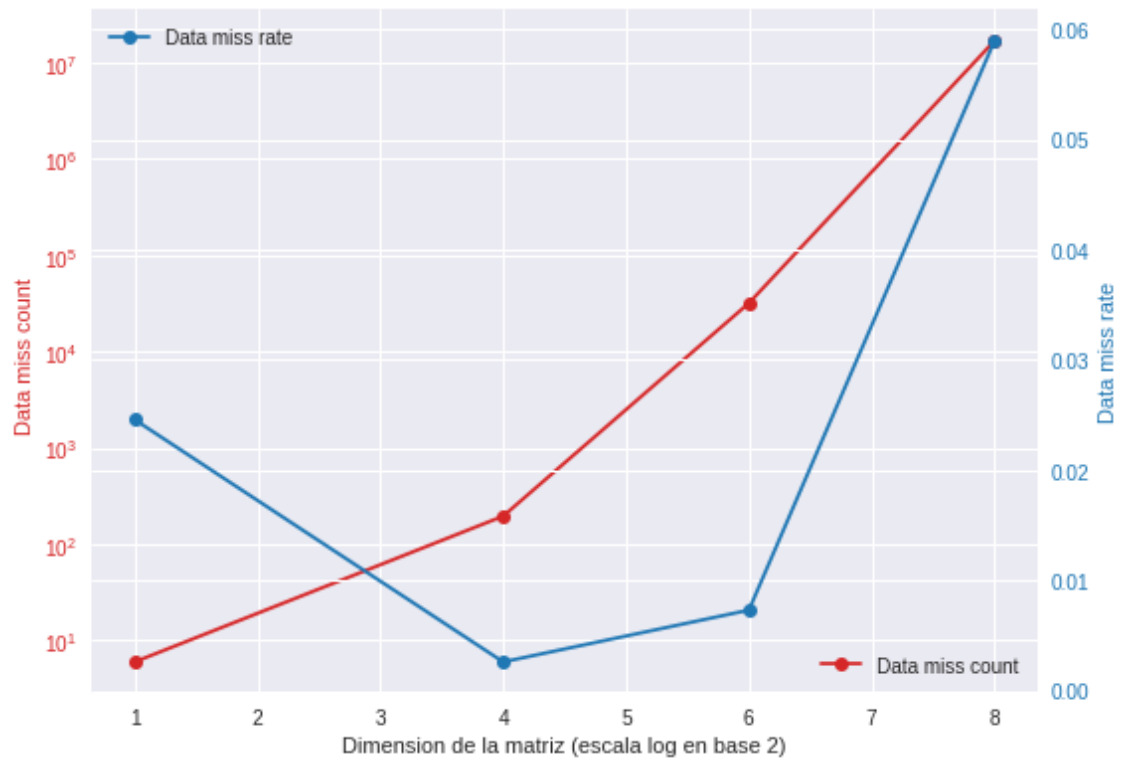


Figura 16: Misses de datos para C3

Como se ve en los gráficos de arriba, la diferencia en cuanto a misses de datos son bastante similares. La mayoría distan de cientos o miles de misses, números que en algunos casos son insignificantes dado que la cantidad de misses totales es bastante superior. Teniendo en cuenta los datos obtenidos anteriormente, se puede llegar a la conclusión de que la diferencia está en el tiempo de ejecución y el rendimiento que puede darse en la realización del proceso del cálculo de la multiplicación de matrices.

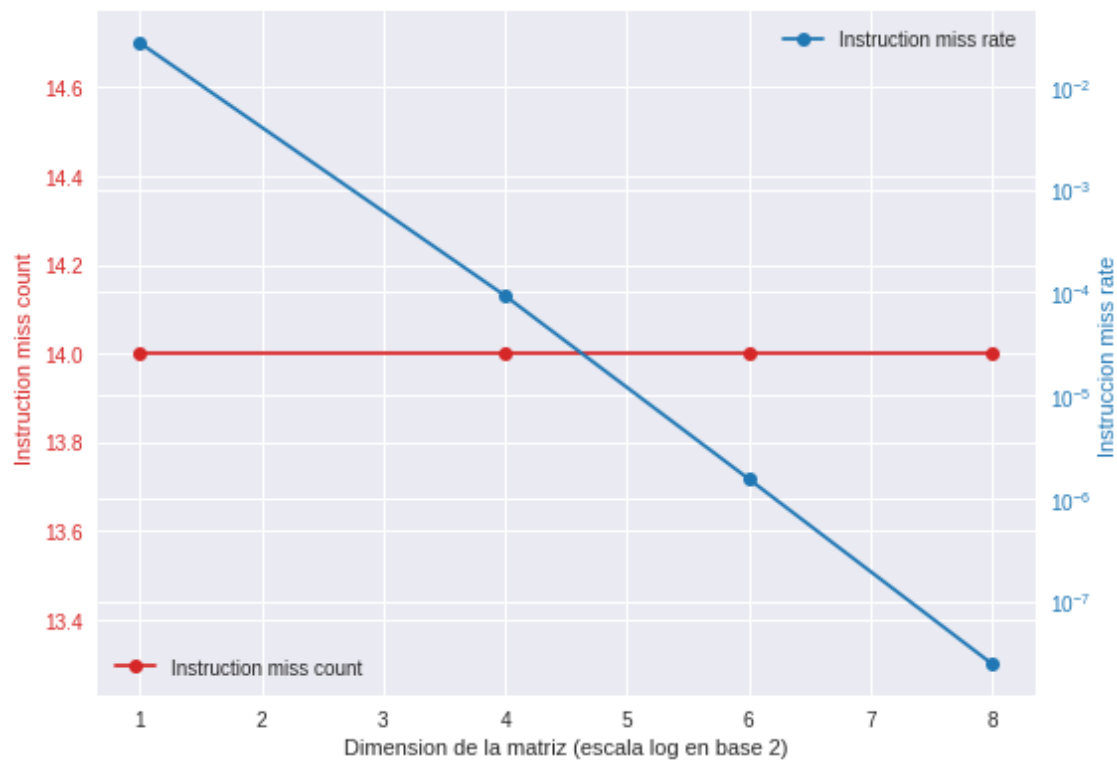


Figura 17: Misses de instrucciones

En cuanto a los misses de instrucciones, se dió lo que se esperaba, que todas las configuraciones de memoria tengan el mismo resultado. Esto se debe a que las instrucciones a leer son siempre las mismas y una cantidad acotada y pequeña de instrucciones.

6. El código fuente de todos los archivos analizados

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "matrix.h"
5
6
7 matrix_t*
8 matrix_multiply(matrix_t* m1, matrix_t* m2, int bs)
9 {
10     size_t n, en, i, j, k, kk, jj;
11     double sum;
12     matrix_t* mr;
13
14     n = m1->rows;
15
16     if(!(mr = create_matrix(n,n))) return NULL;
17
18     en = bs*(n/bs);
19
20     for(i=0; i<n; i++)
21         for(j=0; j<n; j++)
22             mr->array[i*n+j] = 0.0;
23

```

```

24 #if 1
25     if (1) {
26         size_t j;
27         size_t dim = 1024*1024*10;
28         int *v = malloc(dim*sizeof(int));
29         for (j = 0; j < dim; ++j)
30             v[j] = -1;
31         free(v);
32     }
33 #endif
34
35     for(kk=0; kk<en; kk+=bs)
36         for(jj=0; jj<en; jj+=bs)
37             for(i=0; i<n; i++)
38                 for(j=jj; j<jj+bs; j++) {
39                     sum = mr->array[i*n+j];
40                     for(k=kk; k<kk+bs; k++)
41                         sum += m1->array[i*n+k] * m2->array[k*n+j];
42                     mr->array[i*n+j] = sum;
43                 }
44     return mr;
45 }

```

7. El código MIPS32 generado por el compilador

```

1  .section .mdebug.abi32
2  .previous
3  .nan legacy
4  .module fp=xx
5  .module nooddspreg
6  .abicalls
7  .text
8  .align 2
9  .globl matrix_multiply
10 .set nomips16
11 .set nomicromips
12 .ent matrix_multiply
13 .type matrix_multiply, @function
14 matrix_multiply:
15 .frame $fp,88,$31 # vars= 56, regs= 2/0, args= 16, gp= 8
16 .mask 0xc0000000,-4
17 .fmask 0x00000000,0
18 .set noreorder
19 .cpload $25
20 .set nomacro
21 addiu $sp,$sp,-88
22 sw $31,84($sp)
23 sw $fp,80($sp)
24 move $fp,$sp
25 .cpstore 16
26 sw $4,88($fp)
27 sw $5,92($fp)
28 sw $6,96($fp)
29 lw $2,88($fp)
30 lw $2,0($2)
31 sw $2,60($fp)
32 lw $5,60($fp)
33 lw $4,60($fp)
34 lw $2,%call16(create_matrix)($28)
35 move $25,$2
36 .reloc 1f,R_MIPS_JALR,create_matrix
37 1: jalr $25
38 nop
39
40 lw $28,16($fp)
41 sw $2,64($fp)
42 lw $2,64($fp)

```

```

43     bne $2,$0,$L2
44     nop
45
46     move $2,$0
47     b $L3
48     nop
49
50 $L2:
51     lw $2,96($fp)
52     lw $3,60($fp)
53     teq $2,$0,7
54     divu $0,$3,$2
55     mfhi $2
56     mflo $3
57     lw $2,96($fp)
58     mul $2,$3,$2
59     sw $2,68($fp)
60     sw $0,24($fp)
61     b $L4
62     nop
63
64 $L7:
65     sw $0,28($fp)
66     b $L5
67     nop
68
69 $L6:
70     lw $2,64($fp)
71     lw $3,8($2)
72     lw $4,24($fp)
73     lw $2,60($fp)
74     mul $4,$4,$2
75     lw $2,28($fp)
76     addu $2,$4,$2
77     sll $2,$2,3
78     addu $2,$3,$2
79     sw $0,4($2)
80     sw $0,0($2)
81     lw $2,28($fp)
82     addiu $2,$2,1
83     sw $2,28($fp)
84 $L5:
85     lw $3,28($fp)
86     lw $2,60($fp)
87     sltu $2,$3,$2
88     bne $2,$0,$L6
89     nop
90
91     lw $2,24($fp)
92     addiu $2,$2,1
93     sw $2,24($fp)
94 $L4:
95     lw $3,24($fp)
96     lw $2,60($fp)
97     sltu $2,$3,$2
98     bne $2,$0,$L7
99     nop
100
101     li $2,10485760      # 0xa00000
102     sw $2,72($fp)
103     lw $2,72($fp)
104     sll $2,$2,2
105     move $4,$2
106     lw $2,%call16(malloc)($28)
107     move $25,$2
108     .reloc 1f,R_MIPS_JALR,malloc
109 1: jalr $25

```

```

110  nop
111
112  lw  $28,16($fp)
113  sw  $2,76($fp)
114  sw  $0,56($fp)
115  b  $L8
116  nop
117
118  $L9:
119  lw  $2,56($fp)
120  sll $2,$2,2
121  lw  $3,76($fp)
122  addu $2,$3,$2
123  li  $3,-1      # 0xffffffffffffffff
124  sw  $3,0($2)
125  lw  $2,56($fp)
126  addiu $2,$2,1
127  sw  $2,56($fp)
128  $L8:
129  lw  $3,56($fp)
130  lw  $2,72($fp)
131  sltu $2,$3,$2
132  bne $2,$0,$L9
133  nop
134
135  lw  $4,76($fp)
136  lw  $2,%call16(free)($28)
137  move $25,$2
138  .reloc 1f,R_MIPS_JALR,free
139  1: jalr $25
140  nop
141
142  lw  $28,16($fp)
143  sw  $0,36($fp)
144  b  $L10
145  nop
146
147  $L19:
148  sw  $0,40($fp)
149  b  $L11
150  nop
151
152  $L18:
153  sw  $0,24($fp)
154  b  $L12
155  nop
156
157  $L17:
158  lw  $2,40($fp)
159  sw  $2,28($fp)
160  b  $L13
161  nop
162
163  $L16:
164  lw  $2,64($fp)
165  lw  $3,8($2)
166  lw  $4,24($fp)
167  lw  $2,60($fp)
168  mul $4,$4,$2
169  lw  $2,28($fp)
170  addu $2,$4,$2
171  sll $2,$2,3
172  addu $2,$3,$2
173  ldc1 $f0,0($2)
174  sdc1 $f0,48($fp)
175  lw  $2,36($fp)
176  sw  $2,32($fp)

```



```

177     b $L14
178     nop
179
180 $L15:
181     lw  $2,88($fp)
182     lw  $3,8($2)
183     lw  $4,24($fp)
184     lw  $2,60($fp)
185     mul $4,$4,$2
186     lw  $2,32($fp)
187     addu $2,$4,$2
188     sll $2,$2,3
189     addu $2,$3,$2
190     ldc1 $f2,0($2)
191     lw  $2,92($fp)
192     lw  $3,8($2)
193     lw  $4,32($fp)
194     lw  $2,60($fp)
195     mul $4,$4,$2
196     lw  $2,28($fp)
197     addu $2,$4,$2
198     sll $2,$2,3
199     addu $2,$3,$2
200     ldc1 $f0,0($2)
201     mul.d $f0,$f2,$f0
202     ldc1 $f2,48($fp)
203     add.d $f0,$f2,$f0
204     sdc1 $f0,48($fp)
205     lw  $2,32($fp)
206     addiu $2,$2,1
207     sw  $2,32($fp)
208 $L14:
209     lw  $3,96($fp)
210     lw  $2,36($fp)
211     addu $3,$3,$2
212     lw  $2,32($fp)
213     sltu $2,$2,$3
214     bne $2,$0,$L15
215     nop
216
217     lw  $2,64($fp)
218     lw  $3,8($2)
219     lw  $4,24($fp)
220     lw  $2,60($fp)
221     mul $4,$4,$2
222     lw  $2,28($fp)
223     addu $2,$4,$2
224     sll $2,$2,3
225     addu $2,$3,$2
226     ldc1 $f0,48($fp)
227     sdc1 $f0,0($2)
228     lw  $2,28($fp)
229     addiu $2,$2,1
230     sw  $2,28($fp)
231 $L13:
232     lw  $3,96($fp)
233     lw  $2,40($fp)
234     addu $3,$3,$2
235     lw  $2,28($fp)
236     sltu $2,$2,$3
237     bne $2,$0,$L16
238     nop
239
240     lw  $2,24($fp)
241     addiu $2,$2,1
242     sw  $2,24($fp)
243 $L12:

```

```

244 lw $3,24($fp)
245 lw $2,60($fp)
246 sltu $2,$3,$2
247 bne $2,$0,$L17
248 nop
249
250 lw $2,96($fp)
251 lw $3,40($fp)
252 addu $2,$3,$2
253 sw $2,40($fp)
254 $L11:
255 lw $3,40($fp)
256 lw $2,68($fp)
257 sltu $2,$3,$2
258 bne $2,$0,$L18
259 nop
260
261 lw $2,96($fp)
262 lw $3,36($fp)
263 addu $2,$3,$2
264 sw $2,36($fp)
265 $L10:
266 lw $3,36($fp)
267 lw $2,68($fp)
268 sltu $2,$3,$2
269 bne $2,$0,$L19
270 nop
271
272 lw $2,64($fp)
273 $L3:
274 move $sp,$fp
275 lw $31,84($sp)
276 lw $fp,80($sp)
277 addiu $sp,$sp,88
278 jr $31
279 nop
280
281 .set macro
282 .set reorder
283 .end matrix_multiply
284 .size matrix_multiply, .-matrix_multiply
285 .ident "GCC: (Debian 6.3.0-18+deb9u1) 6.3.0 20170516"

```

8. Conclusión

Luego de analizar las diferentes corridas con cachegrind y cg_annotate, pudimos ver que las diferentes maneras de acceder a los datos en la memoria cache modifica significativamente los tiempos de ejecución. No sólo es importante la cantidad de memoria física existente (todos los tipos de cache tienen igual tamaño), sino también es importante la manera de manejarla: cómo almacenar datos y resolver fallos.

Como conclusión, planteamos ventajas y desventajas para cada una de ellas:

- **Memoria cache de mapeo directo** En este caso, cada bloque de memoria cache puede almacenar determinados bloques de memoria principal. Entonces, si hay varias celdas con el mismo número de línea en cache, pero distinto tag el dato va a ser reemplazado cada vez que sea buscado. Esto puede ser considerado como una desventaja debido a la gran probabilidad de que se efectúen varios misses y por consiguiente accesos a memoria, ralentizando así la ejecución del programa. Sin embargo, como ventaja, podemos remarcar su sencilla implementación.
- **Memoria cache asociativa por N conjuntos** Esta implementación, divide la caché en conjuntos de varias líneas. Cada celda puede ir a un único conjunto, dentro del cual puede ir a cualquier línea. La ventaja entonces, es que habrá que borrar un dato de memoria cache

solamente cuando se llene un conjunto, reduciendo así la cantidad de misses con respecto a la caché de mapeo directo.

Para el caso de análisis, llegamos a la conclusión también de que para matrices 2x2 o de tamaño pequeño, la implementación de cache es casi indistinta. Dado que, como se dijo anteriormente, las matrices de este tamaño ocupan muy poco espacio en cache y dejan libres espacios para los datos de las mismas. Entonces, la cantidad de misses y hits será prácticamente igual en cualquier implementacion. Por otro lado, para matrices de tamaño medio o grande, lo ideal seria utilizar el metodo de cache de asociatividad por n vias, ya que este reduce la cantidad de misses del programa y hará mucho mas rápido la ejecución del mismo gracias a su gran flexibilidad.

9. Link al repositorio de github

<https://github.com/manulon/OrgaCompusTP2.git>